June 2001

Adaptive precision setting for cached approximate values

Chris Olston  
*Stanford University*

Boon Thau Loo  
*University of Pennsylvania, boonloo@cis.upenn.edu*

Jennifer Widom  
*Stanford University*

Follow this and additional works at: [http://repository.upenn.edu/cis_papers](http://repository.upenn.edu/cis_papers)

Recommended Citation


Postprint version. Copyright ACM, 2001. This is the author’s version of the work. It is posted here by permission of ACM for your personal use. Not for redistribution. The definitive version was published in *ACM SIGMOD International Conference on Management of Data 2001*, May 2001, pages 355-366. Publisher URL: [http://doi.acm.org/10.1145/376284.375710](http://doi.acm.org/10.1145/376284.375710)

NOTE: At the time of publication, author Boon Thau Loo was affiliated with Stanford University. Currently (April 2007), he is a faculty member in the Department of Computer and Information Science at the University of Pennsylvania.

This paper is posted at ScholarlyCommons. [http://repository.upenn.edu/cis_papers/330](http://repository.upenn.edu/cis_papers/330)

For more information, please contact libraryrepository@pobox.upenn.edu.
Adaptive precision setting for cached approximate values

Abstract
Caching approximate values instead of exact values presents an opportunity for performance gains in exchange for decreased precision. To maximize the performance improvement, cached approximations must be of appropriate precision: approximations that are too precise easily become invalid, requiring frequent refreshing, while overly imprecise approximations are likely to be useless to applications, which must then bypass the cache. We present a parameterized algorithm for adjusting the precision of cached approximations adaptively to achieve the best performance as data values, precision requirements, or workload vary. We consider interval approximations to numeric values but our ideas can be extended to other kinds of data and approximations. Our algorithm strictly generalizes previous adaptive caching algorithms for exact copies: we can set parameters to require that all approximations be exact, in which case our algorithm dynamically chooses whether or not to cache each data value.

We have implemented our algorithm and tested it on synthetic and real-world data. A number of experimental results are reported, showing the effectiveness of our algorithm at maximizing performance, and also showing that in the special case of exact caching our algorithm performs as well as previous algorithms. In cases where bounded imprecision is acceptable, our algorithm easily outperforms previous algorithms for exact caching.

Comments

NOTE: At the time of publication, author Boon Thau Loo was affiliated with Stanford University. Currently (April 2007), he is a faculty member in the Department of Computer and Information Science at the University of Pennsylvania.
Adaptive Precision Setting for Cached Approximate Values*

Chris Olston, Boon Thau Loo, Jennifer Widom
Stanford University
{olson, boonloo, widom}@cs.stanford.edu

Abstract
Caching approximate values instead of exact values presents an opportunity for performance gains in exchange for decreased precision. To maximize the performance improvement, cached approximations must be of appropriate precision: approximations that are too precise easily become invalid, requiring frequent refreshing, while overly imprecise approximations are likely to be useless to applications, which must then bypass the cache. We present a parameterized algorithm for adjusting the precision of cached approximations adaptively to achieve the best performance as data values, precision requirements, or workload vary. We consider interval approximations to numeric values but our ideas can be extended to other kinds of data and approximations. Our algorithm strictly generalizes previous adaptive caching algorithms for exact copies: we can set parameters to require that all approximations be exact, in which case our algorithm dynamically chooses whether or not to cache each data value.

We have implemented our algorithm and tested it on synthetic and real-world data. A number of experimental results are reported, showing the effectiveness of our algorithm at maximizing performance, and also showing that in the special case of exact caching our algorithm performs as well as previous algorithms. In cases where bounded imprecision is acceptable, our algorithm easily outperforms previous algorithms for exact caching.

1 Introduction

Adaptive data caching, e.g., [FC92, WJH97], plays a critical role in the performance of distributed information systems (such as the World-Wide Web) by adjusting the caching strategy dynamically as conditions change. Caching approximate values instead of exact values presents an opportunity for further performance gains in exchange for decreased precision, e.g., [ABGMA88, BGM92, BP93, HSW94, KKO+98, OW00, PL90, WXCJ98]. However, naively choosing what appear to be good approximate values can result in performance gains that are a factor of two (or more) less than the gains achievable when approximate values are chosen carefully, without any improvement in precision. This paper focuses on the problem of setting the precision of cached approximate values dynamically and adaptively to achieve the best possible performance. Now let us motivate the problem somewhat formally, but still abstractly. More concrete details will be given in Section 2 and beyond.

1.1 Approximate Caching and Querying

An approximate data caching environment has one or more data sources, $S_1, \ldots, S_n$. Each $S_i$ hosts a set $V_i$ of exact data values. Each exact value $V \in V_i$ may be cached as an approximation by zero or more caches $C_1, \ldots, C_m$, so each cache $C_j$ may hold an approximation $A_j$ to the exact value $V$. Whenever the value of $V$ changes to $V'$, source $S_i$ applies a boolean test $Valid(A_j, V')$ for each approximation $A_j$ cached by a $C_j$, to decide whether $A_j$ is still a valid approximation for the new value $V'$. If $Valid(A_j, V')$ evaluates to false, the source creates a new approximation $A_j'$ of $V'$ and transmits it to $C_j$ (a value-initiated refresh). Under this protocol, caches are guaranteed to always contain valid approximations, modulo communication overhead.

Typically, not all valid approximations have the same precision, which we will represent by a nonnegative value $Prec(A_j)$. At one extreme, an approximation $A_j$ of $V$ having $Prec(A_j) = \infty$ is an exact copy of $V$. At the other extreme, if $Prec(A_j) = 0$, then $A_j$ contains no information about $V$. In general, an exact value can be approximated with varying degrees of precision between 0 and $\infty$. For example, an integer value could be approximated by an interval, with validity requiring the interval to contain the exact value, and with the width of the interval determining the (inverse of) the precision.

Cached approximate values are accessed by queries running at a cache. If a query finds the precision offered by an approximate value $A_j$ to be insufficient, the query may request the exact value $V$ from the remote source $S_i$. The source responds with the current exact value $V$ as well as a new approximation $A_j'$ to be used by subsequent queries. This exchange is called a query-initiated refresh.

---

* This work was supported by the National Science Foundation under grant IIS-9811947, and by a National Science Foundation graduate research fellowship.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ACM SIGMOD 2001 May 21-24, Santa Barbara, California, USA Copyright 2001 ACM 1-58113-332-4/01/05 ...$5.00.
1.2 Maximizing Performance

Let us assume that our goal is to minimize overall network traffic. Then, in an approximate data caching environment as described in Section 1.1, we want to avoid both value- and query-initiated refreshes as much as possible. The likelihood that either type of refresh will occur depends on the precision of the cached approximation. On one hand, value-initiated refreshes are less likely to occur if the precision is low, since low-precision approximations are likely to remain valid even when the exact value fluctuates to some degree. On the other hand, query-initiated refreshes are less likely to occur if the precision is high, since queries are more likely to find the precision sufficient and not request the exact value from the source. Although the likelihood of each type of refresh can be controlled by adjusting the precision, decreasing the likelihood of one type of refresh increases the likelihood of the other. Therefore, it is not obvious how best to set the precision of cached approximations so as to minimize the overall cost incurred by refreshes of both types.

We will present a parameterized algorithm for adjusting the precision of cached approximations adaptively to achieve the best performance as data values, precision requirements, and/or overall workload vary. The specific problem we address considers interval approximations to numeric values, but our ideas can be extended to other kinds of data and approximations, as discussed briefly in Sections 2.1 and 5. Our algorithm adjusts the precision of each cached approximation independently, and it strictly generalizes previous adaptive caching algorithms for exact copies (e.g., [WJH97]): we can set parameters to require that all approximations be exact, in which case our algorithm dynamically chooses whether or not to cache each data value. We have implemented our algorithm and performed tests over synthetic and real-world data. We report a number of experimental results, which show the effectiveness of our algorithm at maximizing performance, and also show that in the special case of exact caching our algorithm performs as well as previous algorithms. In cases where it is acceptable for queries to produce answers with bounded imprecision, our algorithm easily outperforms previous algorithms for exact caching.

1.3 Related Work

The previous work most similar to ours is Divergence Caching [HSW94], which also considers the problem of setting the precision of approximate values in a caching environment. In their setting, precision is based on number of updates to source values and not on the values themselves—the precision of a cached approximation is inversely proportional to the number of updates since the last cache refresh (referred to as a stale value approximation). The Divergence Caching algorithm proposed in [HSW94] works well in their environment, but does not generalize easily to the kinds of approximations we consider, which are based on the magnitude of source updates instead of their frequency. A more thorough comparison is given in Section 4.7, where we discuss the differences between our work and Divergence Caching in more detail, and show performance results indicating that our approach modestly outperforms Divergence Caching when we specialize ours to the stale value approximations of [HSW94].

No other work that we know of addresses precision setting of cached approximate values while subsuming exact caching techniques. The work on caching approximate values in Interval Relations [BP93], TRAPP [OW00], Epsilon Serializability [PL90], and TACT [YV00] does not focus on the problem of how to set precision optimally. In Quasi-Copies [ABGMA98] and the Demarcation Protocol [BG92], precision cannot be adjusted dynamically. Work on Moving Objects Databases [WCD+98] considers setting precision of cached approximations, but queries are not permitted to request exact values from sources so remote read costs are not taken into account. Conversely, in Soft Caching [KKO98], updates to the exact source value are not considered, so value-initiated refreshes are not considered when setting precision.

1.4 Outline of Paper

The remainder of the paper is structured as follows. In Section 2 we introduce the numeric data values and approximations we consider, and we describe our adaptive precision-setting algorithm. We justify our algorithm mathematically in Section 3. In Section 4 we describe our simulation environment and test data sets, then present our performance results. We first justify empirically our claim from Section 3 that our algorithm converges to optimal performance, by considering steady-state synthetic data. We then switch to real-world data, finding the best parameter settings to maximize the performance of our algorithm under dynamically changing conditions. Next we demonstrate that our algorithm performs as well as previous algorithms in the special case of exact caching, and outperforms exact caching algorithms when exact precision is not required. Finally, we compare our algorithm against Divergence Caching. In Section 5 we conclude and discuss avenues for future work.

2 Setting Interval Precision Adaptively

In this section we describe our algorithm for setting the precision of interval approximations to numeric values. Our ideas can be extended to other types of data and approximations, as discussed briefly at the end of this section. An interval approximation \([I, H]\) is a valid approximation (recall Section 1.1) of a numeric value \(V\), i.e., \(\text{Valid}(I, H, V)\), if \(V\) lies in the interval, i.e., \(I \leq V \leq H\). The precision is the reciprocal of the width of the interval: \(\text{Prec}(I, H) = \frac{1}{H-I}\). At one extreme, a zero-width interval contains only the exact value and thus has infinite precision. In the other extreme, an interval of infinite width gives no information about the exact value and thus has zero precision. We assume that intervals remain
constant until a refresh occurs. Although intervals that vary as a function of time are more general, we have found empirically that they are not particularly helpful, as discussed in Section 4.5.

Whenever the precision \( Prec([L, H]) \) of a cached interval is not adequate for a query running at the cache, i.e., the interval is too wide, the query initiates a refresh by requesting the exact value from the source. (For examples of the kinds of queries that would run over approximate cached values with precision requirements, see, e.g., [OW00, FMP*00].) The source responds with the current exact value as well as a new interval \([L', H']\) to be used by subsequent queries. The cost incurred during a query-initiated refresh will be denoted \( C_{qr} \). A value-initiated refresh incurs cost \( C_{vr} \), and occurs whenever the exact value \( V \) at a data source exceeds its interval \([L, H]\) in some cache, causing \( Valid([L, H], V) \) to become \( false \). Notice that value-initiated refreshes are never required for intervals of infinite width \((H - L = \infty)\). Conversely, when an interval has zero width \((H - L = 0)\), then a value-initiated refresh occurs every time \( V \) changes.

Both types of refreshes (value- and query-initiated) provide an opportunity for the source to adjust the interval being cached. For now let us assume that whenever the source provides a new interval to the cache, the interval is centered around the current exact value. (Uncentered intervals are considered in Section 4.5, and like time-varying intervals they usually turn out not to be helpful.) Therefore, an approximation for a value \( V \) is uniquely determined by the interval width \( W \). The objective in selecting a good interval width is to avoid the need for future refreshes, since we want to minimize communication cost. To avoid value-initiated refreshes, the interval should be wide enough to make it unlikely that modifications to the exact value will exceed the interval. On the other hand, to avoid query-initiated refreshes, the interval should be as narrow as possible. As discussed in Section 1.2, since decreasing the chance of one type of refresh increases the chance of the other, it is not obvious how best to choose an interval width that minimizes the total probability that a refresh will be required.

Both of the factors that affect the choice of interval width—the variation of data values (which causes value-initiated refreshes) and the precision requirements of user queries (which cause query-initiated refreshes)—are difficult to predict, so we propose an adaptive algorithm that adjusts the width \( W \) as conditions change. The overall strategy is as follows. Start with some value for \( W \). Each time a value-initiated refresh occurs (a signal that the interval was too narrow), increase \( W \) when sending the new interval. Conversely, each time a query-initiated refresh occurs (a signal that the interval was too wide), decrease \( W \). This strategy, illustrated in Figure 1, finds a middle ground between very wide intervals that the value never exceeds yet are exceedingly imprecise, and very narrow intervals that are precise but need to be refreshed constantly as the value fluctuates.

Figure 1: Adaptive precision-setting algorithm.

We now define our algorithm precisely. The algorithm relies on five parameters as follows. The first two are functions of the particular distributed caching environment, while the remaining three can be set to tune the algorithm.

(1) the value-initiated refresh cost \( C_{vr} \)
(2) the query-initiated refresh cost \( C_{qr} \)
(3) the adaptivity parameter \( \alpha \geq 0 \)
(4) the lower threshold \( \tau_{0} \geq 0 \)
(5) the upper threshold \( \tau_{\infty} \geq 0 \)

These parameters and others we will introduce later are summarized in Table 1. Let us define a cost factor \( \rho \) as \( \rho = 2 \frac{C_{qr}}{C_{vr}} \). The cost factor is based on the ratio of the two refresh costs and is used to determine how often to grow and shrink the interval width \( W \) as refreshes occur. The mathematical justification for multiplying the ratio by 2 is given in Section 3.

Our algorithm sets the new width \( W' \) for a refreshed interval based on the old width \( W \) during each value- or query-initiated refresh as follows. Recall that \( \alpha \geq 0 \) is the adaptivity parameter.

- value-initiated refresh: with probability \( \min\{\rho, 1\} \), set \( W' \leftarrow W \cdot (1 + \alpha) \)
- query-initiated refresh: with probability \( \min\{\frac{1}{\rho}, 1\} \), set \( W' \leftarrow \frac{W}{(1 + \alpha)} \)

In Section 3 we will justify mathematically why these are the optimal probability settings for width adjustment. Intuitively, the idea is to continually adapt the interval width to balance the likelihood of the two types of refreshes. However, if two value-initiated refreshes are more expensive than one query-initiated refresh, i.e., \( \rho > 1 \), a larger width is preferred, so the width is not decreased on every query-initiated refresh. Conversely, if one query-initiated refresh is more expensive than two value-initiated refreshes, i.e., \( \rho < 1 \), a smaller width
is preferred, so the width is not increased on every value-initiated refresh. Whenever the width is adjusted, the magnitude of the adjustment is controlled by the adaptivity parameter $\alpha$.

Now let us consider the lower and upper thresholds, $\tau_0$ and $\tau_\infty$. When our algorithm computes an interval width $W < \tau_0$, we instead set $W = 0$, and when we compute a $W \geq \tau_\infty$, we instead set $W = \infty$. The purpose of these thresholds is to accommodate boundary cases where either exact caching ($W = 0$) or no caching ($W = \infty$) is appropriate, since without this mechanism the width would never actually reach these extreme values. The source still retains the original width, and uses it when setting the next width $W'$. As part of our performance study we describe how to set these parameters and others.

If a cache does not have enough space to store an approximation for every data value requested, it evicts the widest intervals, since they are the least precise approximations and thus contribute least to overall cache precision. (This decision also is based on original widths, not on 0 or $\infty$ widths due to thresholds.) Caches do not need to notify sources when approximations are evicted. If an evicted approximation incurs a value- or query-initiated refresh, the modified approximation may be cached and another evicted, or the modified approximation may still be the widest and remain uncached.

### 2.1 Other Types of Data and Approximations

We have presented our algorithm specifically for interval approximations to numeric source values, but other types of source data and approximations can use a similar numeric approach. For example, non-numeric data can be approximated by state vectors, where precision is quantified as a numeric measure of the deviation between the exact value and the cached approximation. Then our algorithm can be used to set maximum deviations adaptively in order to optimize overall performance. In Section 4.7 we apply our algorithm in this fashion to emulate Divergence Caching [HSW94], where the deviation metric is the number of updates to the exact value not reflected in the cached approximation. Further exploration of this general topic is left as future work.

### 3 Justification of Algorithm

In this section we justify our algorithm mathematically. Let $P_{vr}$ and $P_{qr}$ represent the probability that a value- or query-initiated refresh (respectively) will occur at each time step. Then the expected cost rate per time step $\Omega = C_{vr} \cdot P_{vr} + C_{qr} \cdot P_{qr}$, where $C_{vr}$ and $C_{qr}$ are the costs of value- and query-initiated refreshes (respectively) as introduced in Section 2.

For a given cached approximation with width $W$, the probability of each type of refresh can be written as $P_{vr} = \frac{C_{vr}}{W^2}$ and $P_{qr} = K_2 \cdot W$, where $K_1$ and $K_2$ are model parameters that depend on the nature of the data and updates, the frequency of queries, and the distribution of query precision requirements. In Appendix A we justify these equations in detail for the case of interval approximations. However, the important—and intuitive—point is that, for all kinds of data

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C_{vr}$</td>
<td>cost of a value-initiated refresh</td>
<td>used to determine cost factor $\rho$</td>
</tr>
<tr>
<td>$C_{qr}$</td>
<td>cost of a query-initiated refresh</td>
<td>used to determine cost factor $\rho$</td>
</tr>
<tr>
<td>$\rho$</td>
<td>cost factor defined as $2 \cdot \frac{C_{vr}}{C_{qr}}$</td>
<td>determines width adjustment probability</td>
</tr>
<tr>
<td>$\Omega$</td>
<td>cost rate (per time step)</td>
<td>metric our algorithm minimizes</td>
</tr>
<tr>
<td>$W$</td>
<td>width of a cached approximation</td>
<td>set adaptively by our algorithm</td>
</tr>
<tr>
<td>$W^*$</td>
<td>width that maximizes the cost rate $\Omega$</td>
<td>our algorithm converges to $W = W^*$</td>
</tr>
<tr>
<td>$\alpha$</td>
<td>adaptivity parameter</td>
<td>how much to adjust width</td>
</tr>
<tr>
<td>$\tau_0$</td>
<td>lower threshold</td>
<td>widths below $\tau_0$ are set to 0</td>
</tr>
<tr>
<td>$\tau_\infty$</td>
<td>upper threshold</td>
<td>widths above $\tau_\infty$ are set to $\infty$</td>
</tr>
<tr>
<td>$P_{vr}$</td>
<td>probability of a value-initiated refresh</td>
<td>increases with precision</td>
</tr>
<tr>
<td>$P_{qr}$</td>
<td>probability of a query-initiated refresh</td>
<td>decreases with precision</td>
</tr>
<tr>
<td>$\delta$</td>
<td>precision constraint of a query</td>
<td>parameter to experiments</td>
</tr>
<tr>
<td>$\Delta\delta$</td>
<td>variation of precision constraints across queries</td>
<td>parameter to experiments</td>
</tr>
<tr>
<td>$\delta_{min}$</td>
<td>minimum precision constraint</td>
<td>derived from $\delta_{max}$ and $\Delta\delta$</td>
</tr>
<tr>
<td>$\delta_{max}$</td>
<td>maximum precision constraint</td>
<td>derived from $\delta_{min}$ and $\Delta\delta$</td>
</tr>
<tr>
<td>$n$</td>
<td>number of data sources</td>
<td>parameter to experiments</td>
</tr>
<tr>
<td>$\kappa$</td>
<td>cache size (in number of approximate values)</td>
<td>parameter to experiments</td>
</tr>
<tr>
<td>$T_q$</td>
<td>time period between queries</td>
<td>parameter to experiments</td>
</tr>
<tr>
<td>$s$</td>
<td>random walk step size</td>
<td>used for analysis in Appendix A</td>
</tr>
</tbody>
</table>

Table 1: Model and algorithm symbols.
and approximations, the value-initiated refresh probability increases with precision (i.e., with a smaller $W$), while the query-initiated refresh probability decreases with precision.

Now that we know how $P_{vr}$ and $P_{qr}$ depend on $W$, we can rewrite our cost rate $\Omega$ in terms of $W$: $\Omega(W) = C_{vr} \cdot \frac{K_r}{W} + C_{qr} \cdot K_2 \cdot W$. Our goal then is to find the value for $W$ that minimizes this expression, which is achieved by finding the root of the derivative. Using this approach, the optimal value for $W$ is $W^* = (2 \cdot \frac{C_{qr}}{C_{vr}} \cdot \frac{K_r}{K_2} \cdot \frac{1}{\rho})$, where $\rho = 2 \cdot \frac{C_{vr}}{C_{qr}}$ is the cost factor we defined in Section 2. Unfortunately, setting the interval width $W$ based on this formula for $W^*$ is difficult unless update behaviors and query/update workloads are stable and known in advance, since model parameters $K_1$ and $K_2$ depend on these factors. One approach is to monitor these factors at run-time to set $K_1$ and $K_2$ appropriately, which is similar to the approach taken by Divergence Caching [HSW94]. However, we will see that our approach achieves the same optimal interval width $W^*$ without the monitoring complexity or overhead.

Our approach is motivated by the observation that $\rho \cdot P_{vr} = P_{qr}$ when $W = W^* = (\rho \cdot \frac{K_r}{K_2} \cdot \frac{1}{\rho})$. Let us first consider the special case where $\rho = 1$. (We will discuss other values for $\rho$ momentarily.) In this special case, the optimal value for $W$ occurs exactly when the two types of refreshes are equally likely. Our algorithm takes advantage of this observation by dynamically adjusting the interval width $W$ so as to equate the likelihood of each type of refresh, thereby discovering the optimal width $W^*$. Our algorithm adjusts the width $W$ based solely on observing refreshes as they occur, without the need for storing history or for direct measurements of updates, queries, or precision requirements. Furthermore, as conditions change over time, our algorithm adapts to always move $W$ toward the optimal width $W^*$. The adaptivity parameter $\alpha \geq 0$, introduced in Section 2, controls how quickly the algorithm is able to adapt to changing conditions.

The graph in Figure 2 illustrates the principle behind our algorithm. Still considering $\rho = 1$, it plots the cost rate $\Omega$ and refresh probabilities $P_{vr}$ and $P_{qr}$ as functions of the interval width $W$. The model parameters $K_1$ and $K_2$ are fixed as $K_1 = 1$ and $K_2 = \frac{1}{100}$. (These values were set based roughly on a query period of 10 seconds and an average precision constraint of 10. Changing these values only shifts the graph.) Notice that the width $W^*$ that minimizes the cost rate $\Omega$ corresponds exactly to the point where the curves for $P_{vr}$ and $P_{qr}$ cross. Therefore, by equalizing the chance that refreshes will be either value- or query-initiated, the optimal width $W^*$ is discovered.

Now consider the general case where $\rho$ can have any value. Our algorithm still discovers the optimal width $W^*$. Recall that the optimal width occurs when $\rho \cdot P_{vr} = P_{qr}$. Our algorithm achieves this condition by not always adjusting the interval width on every refresh. In cases where $\rho < 1$, it is desirable for value-initiated refreshes to be more likely than query-initiated refreshes. Thus, the width is decreased on ev-
We simulated a workload having query period $T_q$ seconds. We will describe how queries and query precision requirements are generated momentarily, but it is important to note that our algorithm is not specialized to any particular type of query. The only assumption made by our algorithm is that for each approximate value, the probability of a query-initiated refresh is proportional to the width of the approximation.

The queries we generate attempt to balance generality and practicality and are inspired by the “bounded aggregate” queries in [OW00]. Each query asks for either the SUM or MAX of a set of approximate values in the cache, where the query result is itself an interval approximation. Each query is accompanied by a precision constraint $\delta \geq 0$ specifying the maximum acceptable width of the result. Precision constraints are generated based on parameters $\delta_{av,g}$ (average precision constraint) and $\Delta \delta$ (precision constraint variation): they are sampled from a uniform distribution between $\delta_{min} = \delta_{av,g} \cdot (1 - \Delta \delta)$ and $\delta_{max} = \delta_{av,g} \cdot (1 + \Delta \delta)$. Using the algorithms in [OW00], from the query type, precision constraint, and approximate data, a (possibly empty) subset of the approximations are selected for query-initiated refresh, after which the desired precision for the query result is guaranteed. Again, it is important to note that our algorithm is not aware of or tuned to these query types or this method of expressing precision requirements—they are used solely to generate a realistic and interesting query load.

4.2 Optimality of Algorithm

Our first experiment was a simple one to verify the correctness of our basic model and the optimality of our algorithm on steady-state data. We used synthetic data consisting of only one source data item, whose value performs a random walk in one dimension: every second, the value either increases or decreases by an amount sampled uniformly from $[0,5,1.5]$. We simulated a workload having query period $T_q = 2$ seconds, average precision constraint $\delta_{av,g} = 20$, and precision constraint variation $\Delta \delta = 1$, in an environment with $\rho = 1$. The query type (SUM or MAX) is irrelevant since we have only one data item.

Our goal was to establish the correctness of our model for the refresh probabilities $P_{vr}$ and $P_{qr}$, i.e., to show that as the data undergoes a random walk $P_{vr}$ and $P_{qr}$ are proportional to $W$ and $W^*$ respectively, and for $\rho = 1$, equalizing $P_{vr}$ and $P_{qr}$ maximizes performance. Thus, we fixed interval width $W$ for each run (i.e., we turned off the part of our algorithm that adjusts widths dynamically), but varied $W$ across runs. We measured the average rate at which value- and query-initiated refreshes occurred, taking their reciprocals to obtain $P_{vr}$ and $P_{qr}$, respectively. Measurements taken during an initial warm-up period were discarded, as in all subsequent reported experiments.

The results are shown in Figure 3, which bears a striking resemblance to Figure 2. The measured values for $P_{vr}$ and $P_{qr}$ are indeed proportional to $\frac{1}{W}$ and $W$, respectively, and the measured cost rate $\Omega$ for different values of $W$ is also shown in Figure 3. From this graph we verify empirically that, in the $\rho = 1$ case, the minimum cost rate does indeed occur when the two refresh probabilities are equal.

We then ran the same experiment letting our algorithm adjust interval widths. The algorithm converged to $W = 3.11$, resulting in performance within 1% of the optimal $W^*$ shown in Figure 3. We further evaluated the optimality of our algorithm with all combinations of $T_q \in [1,2], \delta_{av,g} \in \{10,20\}$, and $\rho \in \{1,4\}$. In all of these scenarios, our algorithm converged to a width resulting in performance within 5% of optimal.

4.3 Our Algorithm in a Dynamic Environment

To test our adaptive algorithm under real-world dynamic conditions, we used publicly available traces of network traffic levels between hosts distributed over a wide area during a two hour period [PF95]. For each host, the data values we use represent a one minute moving window average of network traffic every second, and we picked the 50 most heavily trafficked hosts as our simulated data sources. Traffic levels at these hosts ranged from $0$ to $5.2 \cdot 10^6$ bytes per second. The simulated cache keeps an approximation of the traffic level for at most $\kappa$ of the $n = 50$ sources, where $\kappa$ is a parameter of the algorithm that is set to $\kappa = 50$ unless otherwise stated. Queries are executed at the cache every $T_q$ seconds, computing either the MAX or SUM of traffic over 10 randomly selected sources. In all of our experiments, measurements of the overall cost per unit time $\Omega$ were taken after an initial warm-up period.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{figure3}
\caption{Measured cost rate and refresh probabilities when $\rho = 1$ as functions of interval width.}
\end{figure}
round-trips are required and thus consistency, then \( C_{vr} \) and one response message, so performing a remote read requires one request message. In our experiments, we consider refresh costs \( C_{cr} \) and \( C_{qr} \) that are intended to model network behavior under common consistency models and concurrency control schemes, although our algorithm handles arbitrary cost values. Usually, performing a remote read requires one request message and one response message, so \( C_{qr} = 2 \). If two-phase locking is used for cache consistency, then \( C_{cr} = 4 \) since two round-trips are required and thus \( \rho = 2 \cdot \frac{4}{2} = 4 \). Otherwise, if updates are simply sent to the cache, e.g., in a multiversion or loosely consistent concurrency control scheme, then \( C_{vr} = 1 \) and \( \rho = 2 \cdot \frac{1}{2} = 1 \). Thus, most of our experiments consider the \( \rho \in \{1, 4\} \) cases.

Figures 4 and 5 depict the exact value at one of the data sources for a short segment of a run, along with the cached interval approximation as the value and interval change over time. For illustrative purposes we selected a portion of the run where a host became active after a period of inactivity. These figures illustrate the interval widths selected by our adaptive algorithm with parameters \( \alpha = 1 \), \( \tau_0 = 0 \), and \( \tau_\infty = \infty \), when SUM queries are executed every second (i.e., \( T_q = 1 \)) and \( \rho = 1 \). Figure 4 uses average precision constraint \( \delta_{avg} = 50K \), while Figure 5 uses average precision constraint \( \delta_{avg} = 500K \). When the average precision constraint is small, as in Figure 4, narrow intervals are favored. (To satisfy the precision constraints of SUM queries, the width of each of the 10 individual intervals being summed should be on the order of \( \frac{\delta_{avg}}{10} = \frac{50K}{10} = 5K \) [OW00].) When the average precision constraint is large, as in Figure 5, wide intervals (on the order of \( \frac{\delta_{avg}}{10} = \frac{500K}{10} = 50K \)) are favored.

### 4.4 Setting Parameters

In the remainder of this section we abuse the abbreviation \( K \) for \( \times 10^3 \).

Next, we describe our experiments whose goal was to determine good values for the adaptivity parameter \( \alpha \) and the lower and upper thresholds \( \tau_0 \) and \( \tau_\infty \). First, fixing \( \tau_0 = 0 \) and \( \tau_\infty = \infty \) (meaning we never reset bound widths based on thresholds), we studied the effect of the adaptivity parameter \( \alpha \) on performance. Figure 6 shows the results. We used SUM queries and varied \( \alpha \), considering several different settings for \( T_q \), \( \delta_{min} \), \( \delta_{max} \), and \( \rho \). In this and subsequent experiments, the y-axis cost rate \( \Omega \) is the average for the entire run. All combinations of \( T_q \in \{0.5, 1, 6\} \), \( (\delta_{min}, \delta_{max}) \in \{(50K, 150K), (0, 100K)\} \), and \( \rho \in \{1, 4\} \) are shown. From these experiments and similar experiments using MAX queries, we determined that a good overall set-
the performance degradation is less than \( 1 \) or \( \frac{1}{\Omega} \).

This setting for \( \alpha \) has only a small impact on queries even with moderately large precision constraints. For example, for precision constraints between \( \delta_{\min} = 5K \) and \( \delta_{\max} = 15K \), the performance degradation is less than 1\% (for \( T_q = 1 \), \( \tau_\infty = \infty \), and \( \rho = 1 \)).

Having determined good values for \( \alpha \) and \( \tau_0 \), we now consider the upper threshold \( \tau_\infty \). Setting \( \tau_\infty \) to a small value improves performance for high-precision workloads since it eliminates caching of approximations that are not useful to queries. However, a small \( \tau_\infty \) degrades the performance of low-precision workloads, i.e., those having large precision constraints. To illustrate this tradeoff, in Figures 7, 8, and 9 we plot performance as a function of the average precision constraint \( \delta_{avg} \). Each graph corresponds to a different query period \( T_q \) and shows the performance using three different settings of \( \tau_\infty \), holding all other parameters fixed: \( \rho = 1 \), \( \Delta \delta = 0.5 \), \( \tau_0 = 1K \), and \( \alpha = 1 \). Workloads having \( \delta_{avg} = 0 \) perform best when \( \tau_\infty = \tau_0 \), which guarantees that all intervals are treated as having either no width (\( W = 0 \)) or infinite width (\( W = \infty \)). That is, either the exact value is cached or effectively no value is cached at all. Since queries with \( \delta = 0 \) require exact precision, cached intervals that are not exact are of no use. Note that whenever we set \( \tau_\infty = \tau_0 \), performance is independent of \( \delta_{avg} \) as illustrated by the horizontal lines in Figures 7, 8, and 9. For workloads having a range of different precision constraints, the upper threshold \( \tau_\infty \) should be set to \( \infty \).

Although these guidelines for setting the upper threshold \( \tau_\infty \) apply to most types of queries including our SUM queries, there are exceptions. For example, values can be eliminated as candidates for the exact maximum based on intervals of finite, nonzero width [OW00]. Therefore, for MAX queries, approximate values can be useful to cache even when exact precision is required in all query answers. We have verified experimentally that for MAX queries, setting \( \tau_\infty = \infty \) gives the best performance for all values of \( \delta_{avg} \), including \( \delta_{avg} = 0 \).

In summary, with parameter settings \( \alpha = 1 \), \( \tau_0 = \epsilon \), and \( \tau_\infty = \infty \), our algorithm adaptively selects intervals that
give the best possible performance under dynamically varying conditions. When \( \Delta \delta = 0 \), each query has the same precision constraint, so it is easier for the algorithm to discover the best precision for cached intervals. On the other hand, if \( \Delta \delta \) is large, each query has a different precision constraint, making it harder to find interval widths that work well across multiple queries. Fortunately, it turns out that the degradation in performance due to a wide distribution of precision constraints is small. We verified that the performance of our algorithm is not very sensitive to the precision constraint distribution for several different average precision constraints, while holding the other parameters fixed: \( T_p = 1 \), \( \tau_0 = 1K \), \( \tau_\infty = \infty \), and \( \rho = 1 \). When \( \delta_{avg} = 100K \), the difference in performance between a workload with \( \Delta \delta = 0 \) and \( \Delta \delta = 1 \) is only 1.9%. When \( \delta_{avg} = 10K \), the difference is 5.5%. When \( \delta_{avg} = 5K \), the difference is less than 1%.

4.5 Unsuccessful Variations

We experimented with a number of variations to our algorithm that seemed intuitive but proved unsuccessful in practice: using uncentered intervals, using intervals that vary as functions of time, and adjusting intervals based on the refresh history. We report briefly on our experience with each variation.

An interval is uncentered if, at refresh time, the interval does not bound the exact value symmetrically. Thus, two width values are maintained instead of one: an upper width and a lower width. The source independently adjusts the upper and lower widths as follows. Each time a value-initiated refresh occurs due to the value exceeding the upper and lower widths as follows. Each time a value-val does not bound the exact value symmetrically. Thus, history. We report briefly on our experience with each variation:

an uncentered

4.6 Subsumption of Exact Caching

In this section we compare our algorithm against a state-of-the-art adaptive algorithm for deciding whether to cache exact replicas, which we derive from the replication algorithm in [WJH97]. In this algorithm, the number of requested reads \( r \) and writes \( w \) to each data value are counted. The caching strategy for every data value is reevaluated every \( x \) reads and/or writes to the value, i.e., whenever \( r + w \geq x \). At reevaluation, the projected cost of not caching, i.e., the cost of performing \( r \) remote reads \( C_{nc} = r \cdot C_{or} \) is computed. Similarly, the projected cost of caching, i.e., the cost of performing \( w \) remote writes, \( C_c = w \cdot C_{wr} \) is computed. The value is cached if and only if \( C_c < C_{nc} \). If the cache has limited space, values having the lowest cost difference \( C_{nc} - C_c \) are evicted and the source is notified of the eviction. Under dynamic conditions, it has been shown that this adaptive exact caching strategy continually approaches the optimal strategy [WJH97].

Figures 10, 11, 12, and 13 compare our algorithm against the exact caching algorithm of [WJH97] for SUM queries executed every \( T_q \in \{0.5, 1, 2, 5\} \) seconds. For each run, we first determined the best setting for parameter \( x \) in the exact caching algorithm. Thus we changed the value of \( x \), which varied from 3 to 45, between runs, whereas all of our own parameters remained fixed: \( \alpha = 1 \), \( \tau_0 = 1K \), and \( \tau_\infty \in \{1K, \infty\} \). Figures 10 and 11 show the results for a cache large enough to store all approximate values (\( \kappa = 50 \)), with cost factor \( \rho = 1 \) and \( \rho = 4 \), respectively. Figures 12 and 13 show the results for a small cache of size \( \kappa = 20 \), again with \( \rho = 1 \) and \( \rho = 4 \) respectively.

The performance of our algorithm with \( \tau_\infty = \infty \) almost precisely matches the exact caching algorithm under all workloads, cache sizes, and cost configurations tested. If we set \( \tau_\infty = \infty \), our algorithm offers a significant performance improvement for workloads not requiring exact precision, at
the expense of a slight performance degradation for exact-
precision workloads in the case of SUM queries, as shown
in Figures 10 and 11. When MAX queries are used, our al-
gorithm performs substantially better than exact caching be-
cause values can be eliminated as candidates for the exact
maximum based on cached intervals, as discussed in Sec-
tion 4.4. When the cache size is limited, as in Figures 12
and 13, queries do not benefit much from nonzero preci-
sion constraints because inexact intervals tend to be evicted
from the cache.

4.7 Comparison with Divergence Caching

In Section 1.3 we discussed Divergence Caching [HSW94],
which is the previous work most similar to ours. Recall that in
Divergence Caching, stale value approximations are consid-
ered, where precision is inversely proportional to the number
of updates to the source value not reflected in the cached ap-
proximation, independent of the actual updates. Rather than
adjusting the precision incrementally as our algorithm does,
the Divergence Caching algorithm continually resets the pre-
cision from scratch using detailed projections for data access
and update patterns. These projections are based on past ob-
servations using a moving window scheme where the cache
keeps track of the $k$ most recent reads and the source keeps
track of the $k$ most recent writes. Based on empirical trials,
the window size $k$ was set to 23.

The Divergence Caching algorithm works well in its in-
tended environment, but it is not clear that it could be gener-
alized easily or effectively to incorporate update patterns as
well as frequency. On the other hand, we were able to adapt
our algorithm to handle stale value approximations, and we
report on a preliminary performance comparison. It was a
simple matter to use numeric intervals to bound the number
of updates to the exact source value. We also needed to ad-
just our formula for the cost factor $\rho$ to $\rho' = \frac{c_r}{c_q r}$. Recall that in
our setting, we set $\rho' = \frac{c_r}{c_q r}$ based on a mathematical
analysis (Appendix A). A similar (simplified) analysis of the
value-initiated refresh probability in the Divergence Caching
setting yields $\rho' = \frac{c_r}{c_q r}$. No other modifications to our algo-
rithm were necessary.

Figures 14 and 15 show an initial performance compari-
son of the two approaches. We varied the average precision
constraint $\delta_{av,g}$ (with $\Delta \delta = 1$) for two different query periods $T_q \in \{1, 5\}$. We set costs $C_{vr} = 1$ and $C_{qr} = 2$, giving $p' = 0.5$. For our algorithm we used $\alpha = 1$, $\tau_0 = 1$, and $\tau_\infty = 1$ when $\delta_{av,g} = 0$ and $\tau_\infty = \infty$ otherwise. For the Divergence Caching algorithm we set the window size $k = 23$. We also tried other values for $k$ but no significant performance improvements resulted. As can be seen, our algorithm shows a modest performance improvement over Divergence Caching when we specialize our algorithm to handle stale value approximations.

5 Future Work

In this paper we have considered how to set the precision of approximate values adaptively in dynamic data caching environments. As future work, we plan to investigate adaptive precision setting in symmetric replication architectures, building on work on adaptive exact replication [WJH97] and on replicating interval approximations [YV00]. We also plan to explore algorithms for setting precision in multi-level data caching environments, where each data object resides on one source and there is a hierarchy of caches (e.g., [Ink99]). With multi-level caching, the precision of an approximation in one cache may affect the precision of derived approximations in other caches in the hierarchy. Finally, we plan to investigate how our algorithms and approximations might be applied to other forms of data. For example, environments that cache Web pages could use our approach as discussed in Section 2.1, if the deviation between the exact copy at the source and the stale cached replica can be measured numerically.

Acknowledgments

We thank Mike Franklin for providing useful references regarding exact caching algorithms.

References


A Estimating the Probability of Refresh

To determine the probability of each type of refresh, let us consider a simplified model. First, we model the changing data value as a random walk in one dimension. In the random walk model the value either increases or decreases by a constant amount \( s \) at each time step. A value-initiated refresh occurs when the value moves out of the cached interval approximation. Queries access the data every \( T_q \) time steps. We assume each query accesses only one data value and is accompanied by a precision constraint \( \delta \) sampled from a uniform distribution between \( 0 \) and \( \delta_{\text{max}} \). A query-initiated refresh occurs when the cached interval’s width is larger than a query’s precision constraint \( \delta \). Although this model is simplified, it is useful for deriving formulas and demonstrating the principles behind our algorithm. As we show empirically in Section 4, our algorithm works well for real-world data under a variety of query workloads and precision constraints.

Using our model, we now derive expressions for the value- and query-initiated refresh probabilities at each time step, \( P_{s,r} \) and \( P_{q,r} \) respectively. These probabilities depend on the nature of the data and updates, the frequency of queries, and the distribution of precision constraints. The probability of a query-initiated refresh at a given time step equals the probability \( P_q \) that a query is issued, multiplied by the probability \( P_{\delta < W} \) that the precision of the cached interval does not meet the precision constraint of the query. Clearly, the probability \( P_q \) that a query occurs at each time step is \( P_q = \frac{1}{T_q} \). Recall that in our model, precision constraints are uniformly distributed between \( 0 \) and \( \delta_{\text{max}} \). Thus, as long as \( 0 \leq W \leq \delta_{\text{max}} \), \( P_{\delta < W} = \frac{W}{\delta_{\text{max}}} \). Putting it all together, we have \( P_{q,r} = P_q \cdot P_{\delta < W} = \frac{W}{\delta_{\text{max}}} \). Therefore, the probability of a query-initiated refresh is proportional to the interval width: \( P_{q,r} \propto W \).

Determining a formula for the value-initiated refresh probability requires an analysis of the behavior of a random walk. In the random walk model, after \( t \) steps of size \( s \), the probability distribution of the value is a binomial distribution with variance \( s^2 \cdot t \) [GKP89]. Chebyshev’s Inequality [GKP89] gives an upper bound on the probability \( P \) that the value is beyond any distance \( k \) from the starting point: \( P \leq t \cdot (\frac{k}{s})^2 \). If we let \( k = \frac{W}{\delta} \), and treat the upper bound as a rough approximation, we have the probability \( P_{s,r} \) that the value has exceeded its interval after \( t \) time steps: \( P_{s,r} \approx t \cdot (\frac{W}{\delta})^2 \).

Therefore, the probability of a value-initiated refresh is proportional to the reciprocal of the square of the interval width: \( P_{s,r} \propto \frac{1}{W^2} \).